Janus: A step towards distributed constraint programming

Vijay A. Saraswat, Xerox PARC
Ken Kahn, Xerox PARC
Jacob Levy, Technion, Haifa

(Extended Abstract)

To appear in the
Proceedings of the North American Conference on Logic Programming
Austin, Texas, October 1990

Abstract

We propose a new language, Janus, to address shortcomings of concurrent logic programming languages for distributed programming. Unlike parallel computing, distributed computing must facilitate interactions between very large numbers of diverse computations without central coordination or trust. Janus is a member of the dc family of distributed constraint programming languages, a subset of the cc family [11]. dc languages are "failure-free", which is essential if they are to be used for distributed computing. The general cc framework provides a simple yet formal semantics for Janus, and aids in understanding its relationship with other languages.

Janus, along with other concurrent logic programming languages, is based on asynchronous, buffered message passing and provides, in a simple and clean way, unbounded, recursive invocation of agents, indeterministic selection and "private" communication hidden from the "environment". Its main novelty is its simple yet powerful communication and synchronization mechanism based on single-message, point-to-point, directional channels in which reception or transmission rights for channels can themselves be passed in messages. Janus provides operations upon arrays which enable the expression and efficient execution of conventional programming idioms. It also provides bags of unordered elements which are used for many–to–one communication without the need for merge networks. The language design enables data deallocation and memory reuse as soon as the data becomes garbage. Logical errors that arise during computation can be trapped and handled by user code. Communication protocols between Janus implementations and other computations (possibly other nodes executing Janus) are much simpler than those for other concurrent

logic programming languages which need to support distributed unification and garbage collection, thus enhancing inter-operability.

1 Introduction

We anticipate a future of immense collections of diverse computations using each other without central coordination, trust, or complete knowledge of each other. These *open systems* change dynamically and cannot be centrally administered since they may cross organizational and even national boundaries. While some of the available services may be purely computational, others may interact with the surrounding world.

If this view of the future is to be believed (e.g. see [4],[6]) the question of developing a coherent conceptual framework for organizing such immense collections of computations arises. Some desiderata for such a framework are: it should "scale well", it should avoid all notions of centralized control, its computations should cope with indeterminism and asynchrony of the world and other computations, it should permit definition and installation of securely encapsulated servers with state, its computations should be reprogrammable despite continuous operation, etc.

Traditionally, these issues have been considered at the operating system rather than the language level. We believe that operating systems like Mach [18] provide good support for open systems at a coarse granularity. However, a distributed implementation of a properly designed programming language can provide the same support in a uniform manner at a much finer granularity [7]. A distributed language system consists of some computational nodes all implementing the same language kernel. Entities are referred to in a uniform manner independent of which nodes defined or created them, or where they are "located"; this means that a program developed on a uni-processor will work in a parallel computer or a distributed network of computers. The semantics of such entities is defined in terms of their interaction with their environment which consists of all other concurrently executing entities; if servers offer the same reactive behaviors to their environments, they are equivalent and clients may switch transparently from one to the other. Entities maintain an "arm's length" relationship with each other, allowing the implementation to use locality considerations to improve performance. We believe that only a programming language framework can provide a uniform notation for describing how services are defined and interact in open systems, and prescribe compile-time enforcable rules which ensure that it is impossible to even define certain kinds of undesirable computations.

In this paper we focus on the concurrent logic programming languages [2, 16, 10, 14] because we believe they provide a computation model which may be suitable for open systems programming. A comparison with other frameworks is given in [12]. A general account of concurrent logic programming languages

is given in [11] in the "Ask-and-Tell" cc framework of *concurrent constraint programing languages.* Such languages are fundamentally based on a notion of partial information, together with the notions of consistency and entailment of such information. We treat the (distributed) store as a collection of variables ranging over some domain of interest. Over the life-time of the computation concurrently executing agents may create new variables, and add (*Tell*) constraints relating these variables. Constraints are finitary specifications of possibly infinite sets of values and specify restrictions that any concrete realization of the store must satisfy. A store is *inconsistent* if it is impossible to find a valuation which satisfies the constraints imposed hitherto. Agents synchronize through *a blocking Ask*, by asking whether the store entails another constraint and blocking until—as a result of the execution of other agents—the store entails either the query constraint or its negation. Concurrent logic programming languages such as Parlog, FGHC, *Strand* [15] and Atomic Herbrand [11] (called "FCP(:)" in [9]) are thus concurrent constraint programming languages instantiated over the Herbrand constraint system.

Problems. In the context of distributed computation, a fundamental problem with these languages is that a computation may have multiple producers of information for the same set of variables. Unless these producers are coordinated they may produce conflicting constraints, causing an inconsistency. For example, one agent may impose the constraint `X=3` and another `X=4`, resulting in uncontrolled behavior since an inconsistent store entails any ask condition. This is unacceptable in a distributed setting where aborting the entire networked computation would be disastrous. Indeed, this is the fundamental way in which security of encapsulation of servers can be violated in this computational framework.

Languages such as Atomic Herbrand used atomic publication to avoid the problem: an agent may add constraints to the store only if the resulting store is consistent, and the addition is instantaneous with respect to other attempts. However, in a distributed setting, such a notion is too heavy-weight as a foundational communication mechanism. For Herbrand languages, it requires the implementation of a distributed atomic unification protocol; this additional complexity does not seem motivated for distributed applications and interferes with inter-operability requirements.

Other Herbrand languages such as FGHC ignore the multiple teller problem, trusting the programmer to organize computations so that they will never generate conflicting constraints at run-time. If this trust is violated an inconsistency arises. Mechanisms such as "meta-calls" do not prevent such inconsistencies from occuring and cannot provide any protection if variables are shared across meta-calls.

Distributed constraint programming. For concurrent constraint programming to be used as a conceptual framework for languages for open systems, some restrictions will therefore have to be placed on the use of constraints for communication and control. The principles underlying our approach are:

- Syntactically correct programs should not cause the store to become inconsistent at run-time.

  Consequently constraint-solving is dispensed with at run-time, yielding a style of programming best characterized as "Constraint programming without constraint-solving". The premium is on exploiting the power of *entailment* rather than the power of *consistency*. The role of the run-time system is to put together constraints supplied by various producers and to infer whether some question being asked by a consumer can now be answered.

  Because all possibilities of inconsistency have been removed, notions such as backtracking make no sense, even conceptually.

- The basic communication mechanism should reflect the notion of asynchronous message passing, to be compatible with basic distributed computation concerns.

  Thus computations at a site should be able to progress, producing information, without having to coordinate with agents at other sites.

- Computations in other languages should be able to *inter-operate* smoothly with computations in these languages.

- Languages in this framework should have the potential for performance-efficient implementations competitive with that of conventional (distributed) assignment-based languages.

Specific techniques. We have progressed on two fronts towards these goals.

First, we have identified an efficiently implementable yet expressive constraint system which possesses certain "independence" properties and naturally supports some common idioms in distributed computing.

Suppose `X` and `Y` are two variables unconstrained in the current store, and an agent posts the (consistent) constraint `X=s`, for some (non-variable) term `s` containing `Y`. Then, the constraint system is amenable to a distributed implementation if this constraint imposition *cannot* be construed as a constraint on `Y`. The Herbrand constraint system violates this property, since `s` non-variable and `X=s` implies $X \neq Y$. Therefore, Janus cannot be based on Herbrand, but must instead be based on a constraint system in which `X=s` and `X=Y` are consistent, e.g., the domain of infinite trees.

Other than the issue of independence, our paramount concern has been that the constraint system should be expressive enough to allow elegant formulation of standard algorithms over standard data-structures (for example, arrays) and direct enough that these algorithms should be compilable to efficient code. We

have also discovered that the concept of unordered collections of objects ("bags") is of great use in directly and cleanly supporting many-to-one communication. Hence, Janus is based on a constraint system defined on top of a domain of infinite (possibly integer-indexed) trees and bags.[1]

Second, we have exploited the idea of syntactic control of interference. The intuitive idea of these restrictions is to allow only two occurrences of a variable in a clause (unless the clause forces the variable to be ground). If every clause in the program satisfies this property, and so does the input clause, then it is guaranteed that every resolvent will also satisfy this property (this we called the *broadcast-free* property in earlier work). Thus, (non-ground) variables are regarded as point-to-point communication channels.[2] The second idea, adapated from [5] is to treat these channels as *directional*, by associating with exactly one of the occurrences of the variable (called the *teller* for the variable) the rights to impose constraints on the variable. Surprisingly, directionality can be introduced without introducing any extra-logical annotations which need to be treated specially at run-time (unlike [5] or Concurrent Prolog). Indeed the annotation is "special" only in that clauses must be checked at compile-time to satisfy certain restrictions related to occurrences of the annotation; at run-time the annotation is treated just as any other uninterpreted function-symbol.

Note that it does not make sense to have directional channels and not ban "broadcasting". For, much of the expressiveness of the language arises from the ability to communicate in a message tell rights to a channel, say X, in exactly the same way as other data (e.g., numbers) are communicated. If this message is being communicated to multiple agents (because the variable on which the constraint is being imposed has multiple occurrences), then multiple agents may receive the right to tell constraints on X, thus violating the integrity of the single-teller restriction.

The two occurrence restriction has several ramifications. First, it has the unexpected consequence that cyclic structures created at run-time are always garbage—they cannot be accessed by the rest of the computation, since both occurrences of the variable are used up in creating the cycle. Second, the implementation can detect when the reference to a memory location is no longer valid, since all data has a reference count of one when received by an agent. Hence, it can be reclaimed immediately, without having to incur the overhead of maintaining a reference count (or multiple reference bit). In particular, this means that when an array is modified to a logically new array, the implementation does not need to actually create the new array. Instead, the old array can always be reused and updated in-place. Array sub-sequences can share storage

[1] Bags have to be defined with respect to a non-well-founded set-theory, in order to allow equations like $X = \{X\}$ to have solutions. In particular we use the theory of hypersets developed by Aczel [1]. These technical details are discussed in more depth in [12], but they are not of much practical consequence.

[2] Variables in other logic programming languages such as Prolog or GHC are not point-to-point because of arbitrary fan-out, which may involve multicasting in a distributed setting.

with the original array, and, in some cases, array concatenation can be done in constant time with no additional storage.

In fact, without a garbage collector, a Janus implementation can reclaim garbage as well as a reference count garbage collector. Unreferenced cycles of data or cycles of agents without connections to the main computation can occur and can be reclaimed only by exhaustive garbage collection.

Rest of this paper. Section 2 is devoted to explaining the syntax and semantics of Janus in some more detail and discussing the functionality provided by the underlying constraint system. Section 3 shows that various commonly expressible concurrent logic programming idioms can be expressed straightforwardly in Janus. Section 4 gives a short review of the avenues of research opened up by the design of a practical distributed constraint programming language.

We emphasize that this paper is an extended abstract—for reasons of space, we have been unable to treat many technical details with the attention they deserve. This is repaired in [12].

2 The Janus Language

The syntax used in this paper is a simple variation on the *clausal* syntax common to many concurrent constraint and logic programming languages, and used in [11]. A program is a collection of *clauses*, of the form:[3]

```
behavior_name(ask_term, ..., ask_term) :: agent.
```

where an `agent` is, recursively, a tell-constraint, a simple agent (recursive invocation), a conditional agent (an ask-constraint, followed by an agent), or a parallel composition of agents:

```
agent :: =
      tell_constraint | name(tell_term, ..., tell_term) |
      ask_constraint → agent | agent, agent.
```

This syntax has a few advantages over the more conventional definite clauses. The choice of the syntax "→" for "blocking ask" (conditional agents) is deliberate; this combinator can be construed as a logical entailment operator. It naturally and elegantly allows the specification of nested conditionals thus

[3]Due to shortage of space, we have omitted discussion of some other features of Janus. In fact, a much richer syntax is allowed for Janus programs, a syntax which admits lexically-scoped variables, block-structure, nested definitions (as in [11]) and definitions of (recursive) functions, while preserving a logical interpretation for the syntax.

helping avoid the need for choosing arbitrary procedure names for every branch point in the program.

The operational semantics of Janus is easy to specify. A (non-terminal) configuration consists of a pair $\langle A, \sigma \rangle$, where A is an agent, and σ is the *store*, that is a pair of the form $c : V$, where c is a constraint, and V is the set of variables occurring in A and c. The "telling" of c in a store $c' : V$, terminates in one step, yielding the store $(c \wedge c') : V$. The execution steps of (A_1, A_2) are obtained by interleaving the execution steps of A_1 with those of A_2 in any order. The agent $(c \rightarrow A)$ can, in store σ reduce to the agent A provided that σ entails $\exists X_1, \ldots, X_k.c$ (where $X_1, \ldots, X_k$ are the "local" variables which do not occur in the head of the clause). In this case the store is augmented with an "answer constraint" on the local variables $X_1, \ldots, X_k$ which is conceptually generated by attempting to answer the "query" $\exists X_1, \ldots X_k.c$ given the constraint "database" σ. (Note that because of the presence of bags, there may be more than one answer to any such query. In such a case, any one answer is chosen arbitrarily.)

Similarly, in one step, the agent $p(t_1, \ldots, t_n)$ reduces in a store σ, if there is a clause in the program with head $p(a_1, \ldots, a_n)$ such that the query $\exists Y_1, \ldots Y_k.X_1 = a_1, \ldots, X_n = a_n$ can be answered from the store $\sigma \wedge X_1 = t_1 \ldots X_n = t_n$, where $Y_1, \ldots, Y_k$ are the "local" variables in the query. As for conditional agents, the store is augmented with an answer for the query, on the local variables. Note that more than one clause may be available for reduction—in that case one of them is chosen at run-time, thus yielding the sort of non-determinism generated by Milner's "+" combinator (see [13]). If there is enough information in the store to show that the goal cannot match any clause, then the goal is *removed*. [12] discusses why this is logically defensible—the essential point is to treat the bodies of clauses as certain kinds of implications rather than as conjunctions.

2.1 Tellers

In Janus the term `!X` (called the *teller* for `X`) is used to indicate the right to tell constraints involving `X`. An unannotated occurrence of a variable (e.g., the occurrence in `p(X)`) is called an *asker* for `X`. Legal Janus clauses must satisfy syntactic restrictions that ensure that:

- a variable can occur at most twice in one "thread of execution" of a clause, unless enclosing ask-conditions ensure that the variable will be ground at run-time,
- at most one of those occurrences (called the *source*) is an occurrence through which bindings can be generated or can flow in from outside the clause, and at most one of those occurrences (called the *sink*) is an occurrence through which bindings are checked or can flow out of the clause,

- in particular, if the body of a clause imposes a constraint on a variable X that occurs in the head of a clause, then the occcurrence in the head must be the teller for X (so that at run-time the goal will be checked to ensure that it has tell rights on X). Here the occurrence in the body is the source and that in the head, the sink.

Thus the teller for `X` represents the capability to impose (tell) constraints on `X`, and the asker for `X` the capability to read (ask) constraints on `X`. Very importantly, it is possible in Janus to *communicate* these capabilities in messages to other agents. A detailed account of a simple, liberal set of syntactic rules faithful to these intuitions is given in [12]; all programs in this paper satisfy those restrictions, and informally can be seen to satisfy these intuitions.

As a simple example, consider append expressed in Janus:

```
append([], In, !Out) :: Out = In.
append([A | B], In, !Out) :: Out = [A | Out1], append(B, In, !Out1).
```

The clauses obey the Janus restrictions. Intuitively, in the first clause, the first occurrence of `Out` is the sink and the second the source; and the first occurrence of `In` is the source and the second the sink. In the second clause, intuitively, the first occurrences of `A,B,In` and the second occurrences of $\texttt{Out}, \texttt{Out}_1$ are sources. Operationally, a call `append(X,Y,Z)` will suspend until the store has some information on `X` and `Z`. The first clause can be taken if the store implies `X=[]` and `Z=!S` for some term `S`[4]; and the second clause can be taken if the store implies `X=[C | D]` (for some terms `C` and `D`), and `Z=!S`, for some variable `S`. In case both of these clauses are rejected (e.g., because the store entails `X=3` or `Z=[]`), the call terminates, without any constraint being added to the store. Intuitively, such a program defines the behavior of append *only* in those circumstances in which the first argument of the call is a well-formed list, and the last argument is a teller.

2.2 Constraint system

The domain of the constraint system $\mathcal{J}$ underlying Janus consists of a set of constants ("urelements"; these include arithmetic constants) and, recursively, possibly infinite ordered and unordered collections (bags) of such elements. Ask and Tell constraints are interpreted over this domain. The permissible ask constraints include:

- `X` = `E` where `E` is any expression.
- Various type and arithmetic tests such as `integer(X)`, `X` $\geq$ `Y`, etc.

[4] It is a property of Janus that in any store generated at run-time, terms such as `S` will be variables.

- The array operations $|\mathtt{A}|$ to obtain the cardinality of A and A.I which obtains the ith element of the array A. These operations also implicitly ask array(A), integer(I) and $0 \leq \mathtt{I} \leq |\mathtt{A}| - 1$ (that is, they suspend until the implicit ask conditions can be answered by the store).

- In addition, terms of the form $\{\mathtt{M}\} \cup \mathtt{B}$ may occur in the head of a clause, provided that M and B are variables that do not occur elsewhere in the head of the clause.

  Intuitively, an invocation can "match" a clause-head which contains the term $\{\mathtt{M}\} \cup \mathtt{B}$, if its corresponding argument is known to be a non-empty bag (for example, it may be a non-empty enumerated bag, or a variable A on which a constraint of the form $\mathtt{X_1} \cup \ldots \cup \mathtt{X_n} = \mathtt{A}$ has been imposed and at least one of the $\mathtt{X_i}$ is an enumerated bag, see below). The match succeeds by equating M to one of the elements in the bag, and B to the "rest" of the bag. (Formally, all this happens in the attempt to answer the query from the constraint database.)

The only tell constraints permitted in Janus are of the form $\mathtt{X} = \mathtt{E}$ where E can be any expression including arithmetic, array, and bag expressions.

Bag expressions are either of the form $\{\mathtt{T_1}, \ldots, \mathtt{T_n}\}$ or $\mathtt{X_1} \cup \ldots \cup \mathtt{X_n}$. The former denotes the bag that contains n elements, $\mathtt{T_1}, \ldots, \mathtt{T_n}$, and the latter denotes the bag union of the $\mathtt{X_i}$s. For convenience, union is defined in such a way that if any of the $\mathtt{X_i}$ is not a bag it is treated as the singleton bag $\{\mathtt{X_i}\}$.

Array expressions are of the form:

- $\langle \mathtt{T_1}, \ldots, \mathtt{T_n} \rangle$, an array of exactly n elements, $\mathtt{T_i}$ being the ith element,

- array(N, V), an array of N Vs. (This implicitly asks that N is a non-negative integer and V is forced (i.e. ground).)

- A[I..J] the subarray of A between the indices I and J (this implictly asks that I and J be integers in the proper ranges)

- $\mathtt{A_1} \circ \mathtt{A_2}$, an array which is the concatenation of $\mathtt{A_1}$ and $\mathtt{A_2}$.

- $\mathtt{A}[\mathtt{I_1} \rightarrow \mathtt{T_1}, \ldots, \mathtt{I_n} \rightarrow \mathtt{T_n}]$, the array different from A only in that each index $\mathtt{I_j}$ has value $\mathtt{T_j}$.

Further details related to $\mathcal{J}$, and the semantics of Janus are contained in [12]. There we prove, among other results, that Janus computations are failure-free, and that variables serve as point-to-point communication channels.

3 Programming in Janus

We lack the space to adequately demonstrate the richness of programming idioms available to the Janus programmer. Instead, we give several examples

which we think are representative of various aspects of the language. First we present how to express commonly used concurrent logic programming techniques in Janus. The need to specify a directionality for communication and the restrictions on the ability to share data lead to only small changes in most of the common programming techniques, such as bounded-buffer, incomplete messages, difference lists, short circuits, recursive doubling, buffer synchronization etc.

3.1 Common programming idioms adapted to Janus

The standard bounded-buffer producer-consumer idiom can be programmed in Janus as follows:

```
goal :: producer(!Bs, [demand, demand, ..., demand | Ds]), consumer(Bs, !Ds).
producer(!Bs, Ds) ::
        Ds = [demand | Ds1] → Bs = [produced | Bs1], producer(!Bs1, Ds1),
        Ds = [] → Bs = [].
consumer(Bs, !Ds) ::
        Bs = [produced | Bs1] → Ds = [demand | Ds1], consumer(Bs1, !Ds1),
        Bs = [] → Ds = [].
```

(Note that the `producer` goal is given the teller for `Bs`; this is checked in the head of the clause for `producer`, whose body generates constraints on `Bs`. Similarly for `Ds`.) This producer-consumer pair executes with a bounded buffer between the producer and consumer. *It is very important to note that in a* Janus *implementation that reuses structures, the above program executes in* constant space *if both goals reside in the same address space.* This is possible because each clause in the program is "memory-balanced", that is, each "consumes" as many conses as it "produces"; hence the conses introduced in the initial query are *recycled* for the duration of the computation.

The short-circuit technique can be programmed in Janus as follows:

```
p(..., L, R) :: p(..., L, !M), p(..., M, R).
p(..., L, !R) :: R = L.
```

In the first clause, the first occurrence of `M` is the source and the second the sink. Here, the programmer has to impose a consistent directionality of information flow. An initial query might be `p(done,!D),q(D)`. `q` can wait until `D=done` succeeds, before proceeding. A good implementation of Janus can exploit the directionality of these short-circuits so that it is never the case that `D` is bound to `done` via a long dereference chain.[5]

[5] A more detailed exposition, which handles extensions of the usual short-circuit protocol is given in [12].

3.2 Incomplete Messages

A major source of simplicity and elegance in concurrent logic programming is that servers can respond directly to service requests by communicating on ports embedded in the service request. This technique is illustrated by the following "queue-server" program:

```
queue(Input) :: split(Input, !Enq, !Deq), match(Enq, Deq).
split(enqueue(X, Rest), !Enq, Deq) :: Enq = [X | Enq1], split(Rest, !Enq1, Deq).
split(dequeue(!X, Rest), Enq, !Deq) :: Deq = [!X | Deq1], split(Rest, Enq, !Deq1).
match([X | Enq], [!Y | Deq]) :: Y = X, match(Enq, Deq).
```

These rules define a predicate `queue/1`, which will receive a sequence of `enqueue` and `dequeue` messages over time and will be able to respond as if these operations were being performed on a queue.[6] Operationally, a `queue` agent splits into two agents. One monitors the input stream and distributes it into two queues, one consisting of the enqueue requests received (preserving order of arrival) and the other consisting of dequeue requests received. Each enqueue request contains the value to be enqueued. Each dequeue request contains a teller for a variable, the asker for which is retained by the agent sending the request. When an enqueue request arrives, it is communicated via the tell capability directly to the agent which requested that the dequeue operation be performed. (Note that the length of the dequeue list can be greater than the length of the enqueue list, so that the queue may be "negative" for a while.) If we wish to give clients separate "enqueue" and "dequeue" capabilities then only the `match/2` program is needed.

3.3 One to many communication

There are several programming techniques which compensate for the inability to directly share data which is a result of the two-occurrence restriction. As we now show (and as was partly shown in [5]), it is very easy to write a program which can copy the data on one channel into any number of outgoing channels, even if the incoming data contains a teller.

The `distribute` predicate takes `n+2` arguments. The first argument is the channel on which communication is received, and the rest are channels to which the communication is to be copied. (One of these channels, the second argument, is treated specially.) The basic idea is very simple. We illustrate it for the case in which an agent can receive only arrays of length 2.

[6] Typically the input stream is obtained by merging one stream from each client—the handling of many-to-one communication schemes is discussed in more detail below.

```
distribute([X1 | X2], !Y0, ..., !Yn) ::
        Y0 = [Y01 | Y02], ..., Yn = [Yn1 | Yn2],
        distribute(X1, !Y01, ..., !Yn1), distribute(X2, !Y02, ..., !Yn2).
```

However, in addition, it is possible that the agent receives the teller for another stream on its input channel. In this case, we would like to forward this teller to some *one* of the "output" streams, but in such a way that we may continue to monitor the output produced on this channel and copy it to the other streams. To achieve this, we send the teller for a *new* communication channel `Z` to the selected stream, and then recur with `Z` as the new "input" stream for the distributor. In this way the role of the asker and teller may ping-pong between the first two arguments of the `distribute` agent:

```
distribute(!X, !Y0, ..., !Yn) :: Y0 = !Z, distribute(Z, !X, !Y1, ..., !Yn).
```

For many programs, explicit calls to a `distribute` predicate can be eliminated in favor of in-line unfolding of the above code.

Other techniques for sharing data are also possible. Frequently the data structure can be passed in a pipelined fashion to avoid all the overhead of copying. Access to the data can, alternatively, be serialized so that only one agent at a time has access. Finally, the data can be made the local state of an agent which has many clients that query the data structure. These issues are further discussed in [12].

3.4 Janus programs using arrays

A major use of arrays is as a random-access data-structure internal to a process, which can be updated sequentially. This is illustrated by the following program for the "Dutch National Flag" problem [3]:

```
dnf(Cols, Sorted) :: dnf(Cols, 0, |Cols| − 1, |Cols| − 1, Sorted).
dnf(In, R, W, B, !Out) ::
            R > W → Out = In,
            R ≤ W → (In.W = red → dnf(In[R → In.W, W → In.R], R + 1, W, B, !Out),
                     In.W = white → dnf(In, R, W − 1, B, !Out),
                     In.W = blue → dnf(In[B → In.W, W → In.B], R, W − 1, B − 1, !Out)).
```

The above program is almost a direct translation of the sequential program in [3]. Note the idiom for swapping two elements of the array, and recall that assignments can be done in place. The code produced by a Janus compiler for this program for a uni-processor machine is not expected to be very different

from code produced for the corresponding program in an imperative, sequential language.

In Janus it is possible to quicksort an array in parallel *in place*:

```
qsort(A, !B) ::
        |A| ≤ 1 → B = A,
        |A| > 1 → split(A, !K, !E), qsort2(E, K, !B).
```

The `split(A, !K, !Result)` agent pivots `A` around `A.0`, returning the elements of `A` less than `A.0` in `Result`$[0..(K-1)]$, `A.0` in `Result.K`, and elements of `A` not less than `A.0` in `Result`$[(K+1)..|A|]$..

```
split(A, K, Result) :: split(A, 1, |A| − 1, K, Result).
split(A, I, Bound, !K, !Result) ::
        (I < Bound →
            (A.0 > A.I → split(A, I + 1, Bound, !K, !Result),
             A.0 ≤ A.I → split(A[Bound → A.I, I → A.Bound], I, Bound − 1, !K, !Result)),
         I = Bound →
            (A.0 > A.I → Result = A[0 → A.I, I → A.0], K = I,
             A.0 ≤ A.I → Result = A[0 → A.(I − 1), (I − 1) → A.0], K = I − 1)).

qsort2(E, K, !B) ::   %Implicitly suspends until E is an array
        0 < K, K < |E| − 1 → qsort(E[0..K], !C), qsort(E[(K + 1)..(|E| − 1)], !D), B = C ∘ D,
        K = 0 → B = E,
        K = |E| − 1 → B = E.
```

Essentially, `split` pivots its input array `A` around `A.0`, in place. The two halves of the array can then be sorted recursively in place, and the results combined in place in constant time since the pieces are contiguous. To our knowledge, it is not possible to obtain such a simple and efficient quick-sort program in other concurrent logic programming languages.

3.5 Bags

The communication mechanism discussed so far has focussed on point-to-point directional communication channels. An agent may have a fixed number of such channels. Such communication schemes are too simplistic, however, to deal elegantly with many to one communication. Consider for example a situation in which an a priori unbounded number of clients may seek service from a server. Many-to-one communication can be achieved in Janus using *bags* which are essentially (possibly infinite) unordered collections.

The reader of a bag suspends waiting for an element to appear in the bag. It then processes that element and typically recurs with the remaining bag.

Each writer of the bag B has tell rights to a variable X_i and $B = X_1 \cup \ldots \cup X_n$ has been told to the store. A writer can split its reference m ways by telling $X_i = X_{i1} \cup \ldots X_{im}$. A writer can drop its tell rights by imposing the constraint $X_i = \{\}$.

A simple bank account program is an example of a typical use of bags:

```
account({M} ∪ In, Balance) ::
        M = deposit(Amount), Amount ≥ 0 → account(In, Balance + Amount),
        M = withdraw(Amount, !Ack) →
                ((0 ≤ Amount, Amount ≤ Balance) → Ack = ok, account(In, Balance − Amount),
                 Amount > Balance → Ack = overdraw_attempt(Amount), account(In, Balance),
                 Amount < 0 → Ack = negative_withdrawal(Amount), account(In, Balance))
        M = current_balance(!Reply), number(Balance) → Reply = Balance, account(In, Balance).
```

The guard `number(Balance)` was added to the last behavior of `account` to indicate that `Balance` is forced and can therefore occur multiple times. Straightforward flow-analysis-based compilation techniques can lead to the elimination of this test at run-time.

Note that it is possible to use bags to obtain many-to-one *sequenced* communication: this is done by the client sending to the server a list (array of two elements). The server processes the first element in the list, and places the tail back in the bag. This guarantees that elements in each list are processed in the order in which they occur in the list.

Bags provide a more general and less ad-hoc alternative to the primitive merge servers found in many other concurrent logic programming languages. Bags can support message sending in which messages are unordered as well as those in which the relative order of sends from each client is preserved. Unlike bags, primitive merges force servers to deal with lists of items and clients to follow a system-wide protocol for stream splitting. Rather than provide primitive implementations for a particular library agent, bags provide a more principled and general foundation for accomplishing the same goals.

4 Comparison with Other Work and Conclusions

Our work on distributed constraint programming languages was influenced by [5]. From [5] we learnt how to combine the idea of *broadcast-freedom* with a static annotation scheme. Judiciously defined, the annotation scheme turned out to be enough to obtain failure-freeness, besides having a very intuitive operational flavor. The main difference between Doc and Janus is that Doc programs are not failure-free—even ignoring the "occurs-check" problem. A goal is allowed

to reduce without checking that the basic operations in the clause can be executed successfully. In particular, a goal is allowed to reduce without checking that teller rights have been obtained for all the variables on the left hand side of equality goals: hence at run-time, it is possible situations such as `2=3` may occur, requiring the computation to abort. Another major difference is that the annotation scheme in Doc was rather complex and was presented as a modification to the unification algorithm. (Doc annotated every variable with either a "`<`" annotation or an "`>`" annotation. Consequently, the most general unification algorithm had to be modified to take into account the proper rules for matching "`X<`" with "`Y>`".) Our contribution here has been to present essentially the same idea (using the notion of tellers) *without* having to modify the unification algorithm at all.[7]

Our development has also made it possible to extend the notion of askers and tellers to constraint systems far richer than Herbrand, namely systems with rich functionality for arrays and bags. Also to our knowledge, Doc was never seriously implemented or intended as a language which would be competitive with concurrent imperative languages.

Recently, certain uniprocessor implementation schemes for FGHC have been described [17], for a subset of FGHC similar to Janus. These schemes may have some of the potential for generating tight code that Janus programs have. An advantage of writing code in Janus is that, while the programmer must provide more knowledge about the directionality of communication, this knowledge can be exploited by the compiler. We are skeptical that on comparable code as good a job can be done by a compiler for FGHC as by one for Janus. Also, the concepts of distributed programming match Janus much more closely than other `cc/Herbrand` languages based on run-time constraint solving.

Most Janus programs can be written directly as *Strand* programs. However, a *Strand* program does not have the guarantee that it will not abort at run-time due to "assignment errors". Our work here can be seen as devising liberal compile-time checkable rules that ensure that no "assignment errors" will occur at run-time. In a distributed programming language, it is important that the compiler be able to guarantee that no run-time errors caused by a careless programmer can compromise the integrity of network-wide computations, and this cannot be done in Strand, without restricting the user to a language essentially like Janus.

Future work. The work we describe above is part of a project to investigate language-level support for constructing immense distributed systems. In the context of this project, we wish to examine a set of issues and their interactions with language design, including the design of language extensions for input/output, the development of strong typing schemes for the language,

[7] In fact, Janus restrictions imply that unification in Janus can be compiled into simple pattern-matching code.

development of an adequate vocabulary for explicit run-time reconfigurability, the development of "fail-safe" implementations that can survive node-failures and network partitions, the implementation of secure implementations which are immune from malicious attempts to compromise the integrity of a node, the development of semantically-based program design, development and debugging tools etc. We believe it will be possible to fruitfully address many of these tasks within this framework because the design of Janus seems to be as simple as it can be, without compromising the usability of the language for distributed computation.

Conclusion. We have introduced a simple distributed programming language based exclusively on the use of constraints for communication and control. Thus the language has a clean logical foundation and a very simple formal semantics (e.g., see [13]). It extends prior concurrent logic programming languages in that computations are guaranteed not to result in inconsistent stores and in that the underlying constraint system is substantially richer than that found in conventional logic programming languages.[8] The restrictions imposed in achieving the failure-freedom property naturally reflect the view of variables as single-message, point-to-point *directional* communication channels, and naturally allow the communication of tell- and ask-rights to channels in messages. At the same time, it seems likely that the language can be implemented so as to be competitive with conventional, distributed assignment-based languages. Indeed, we see this work as being a step on the way towards the establishment of a new paradigm for distributed computing, the distributed constraint programming paradigm.

Acknowledgements. These ideas have been discussed with a number of people over several months. The research underlying this paper has benefitted tremendously from many, many discussions with Mark Miller and Eric Dean Tribble. We also thank Danny Bobrow, Seif Haridi, Fernando Pereira, Julie Basu, Saumya Debray, Mats Carlsson, Volker Haarslev, Bogumil Hausman and David H.D. Warren for discussions and comments.

References

[1] Peter Aczel. *Non well-founded sets.* Number 14 in CSLI Lecture Notes. CSLI, 1988.

[2] K. L. Clark and S. Gregory. Parlog: parallel programming in logic. *TOPLAS*, 8(1):1–49, January 1986.

[3] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[8] In particular, Janus does not make use of *any* non-logical aspects other than those embodied in the notion of blocking Ask; this in contrast with almost all other concurrent logic programming languages which either introduce gratuitously non-logical operations (Concurrent Prolog, the assignment in Strand), or fail to specify the built-in primitives in enough detail (e.g. GHC).

[4] Carl Hewitt. *The Ecology of Computation*, chapter Offices are open systems. Elsevier Science Publishers/North-Holland, 1988.

[5] Masahiro Hirata. Programming language Doc and its self-description, or, `X = X` considered harmful. In *3d Conference Proceedings of Japan Society for Software Science and Technology*, pages 69–72, 1986.

[6] Bernardo Huberman, editor. *The Ecology of Computation.* Elsevier Science Publishers/North-Holland, 1988.

[7] Ken Kahn and Mark S. Miller. *The Ecology of Computation*, chapter Language Design and Open Systems. North Holland, 1988.

[8] Kenneth M. Kahn and Vijay A. Saraswat. Complete visualization of concurrent programs and their execution. In *Proceedings of the IEEE Workshop on Visual Programming.* IEEE, October 1990. Also SSL technical report, Xerox PARC, August 1989.

[9] S. Kliger, E. Yardeni, E. Shapiro, and K. Kahn. The language **FCP(:,?)**. In *Conference on Fifth Generation Computer Systems*, December 1988.

[10] Vijay A. Saraswat. The concurrent logic programming language **CP**: definition and operational semantics. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 49–62. ACM, January 1987.

[11] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989. To appear, Doctoral Dissertation Award and Logic Programming Series, MIT Press, 1990.

[12] Vijay A. Saraswat, Kenneth M. Kahn, and Jacob Levy. Distributed constraint programming—the dc framework and Janus. Technical report, Xerox PARC, August 1989.

[13] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages, San Francisco*, January 1990.

[14] Ehud Shapiro, editor. *Concurrent Prolog, Volume 1 and 2.* MIT Press, December 1987.

[15] Strand Software Technologies Inc. *Strand 88 Users Manual*, June 1989.

[16] K. Ueda. *Guarded Horn Clauses.* PhD thesis, University of Tokyo, 1986.

[17] K. Ueda and M. Morita A new implementation technique for Flat GHC. In *Proceedings of the Seventh International Conference on Logic Programming*, June 1990.

[18] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and D. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh Symposium on Operating System Principles*, November 1987.